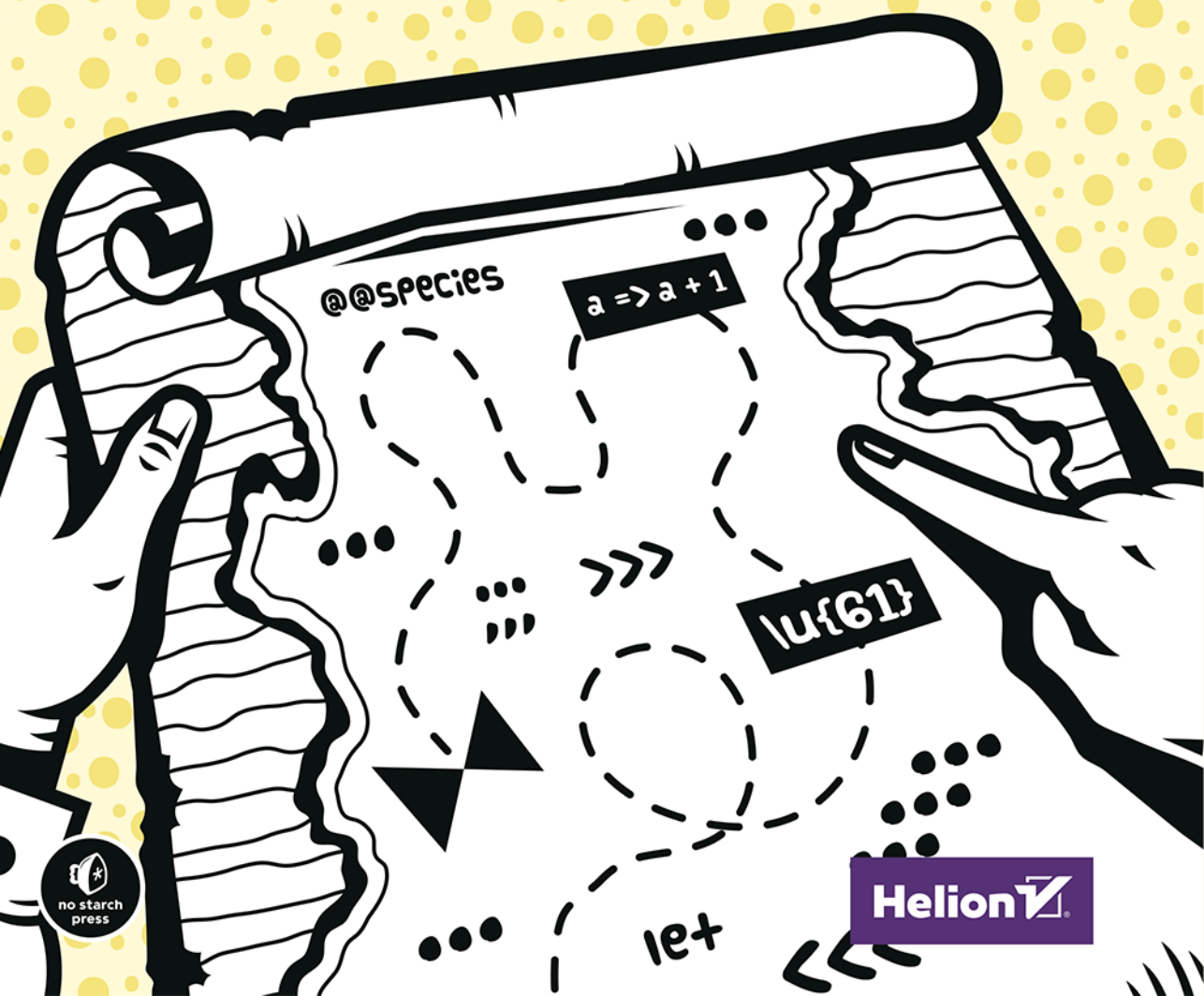


ECMAScript 6

PRZEWODNIK PO NOWYM
STANDARDZIE JĘZYKA JAVASCRIPT

NICHOLAS C. ZAKAS



Tytuł oryginału: Understanding ECMAScript 6: The Definitive Guide for
JavaScript Developers

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-3403-8

Copyright © 2016 by Nicholas C. Zakas

Title of English-language original: Understanding ECMAScript 6,
ISBN 978-1-59327-757-4, published by No Starch Press.

Polish-language edition copyright © 2017 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/ecmasc>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O AUTORZE	11
WSTĘP	13
WPROWADZENIE	15
Droga do ECMAScript 6	15
O tej książce	16
Zgodność przeglądarki WWW i Node.js	17
Dla kogo przeznaczona jest ta książka?	17
Ogólne omówienie zawartości książki	17
Konwencje typograficzne użyte w książce	19
Pomoc i wsparcie	20
I	
WIĄZANIE BLOKÓW	21
Deklaracje var i hoisting	21
Deklaracje na poziomie bloku	23
Deklaracje let	23
Brak możliwości ponownej deklaracji	24
Deklaracja const	25
Tymczasowo martwa strefa	27
Wiązanie bloku w pętli	28
Funkcja w pętli	29
Deklaracja let w pętli	30
Deklaracja const w pętli	31
Wiązanie bloku globalnego	32
Zastosowanie najlepszych praktyk dotyczących wiązania bloku	33
Podsumowanie	34

2

CIĄGI TEKSTOWE I WYRAŻENIA REGULARNE 35

Lepsza obsługa Unicode	35
Punkty kodowe UTF-16	36
Metoda <code>codePointAt()</code>	37
Metoda <code>String.fromCodePoint()</code>	38
Metoda <code>normalize()</code>	38
Opcja <code>u</code> dla wyrażeń regularnych	40
Inne zmiany dotyczące ciągu tekstowego	42
Metody przeznaczone do identyfikacji podciągów tekstowych	42
Metoda <code>repeat()</code>	44
Zmiany wprowadzone w wyrażeniach regularnych	45
Opcja <code>y</code> w wyrażeniach regularnych	45
Powielanie wyrażenia regularnego	48
Właściwość <code>flags</code>	49
Szablony literałów	50
Składnia podstawowa	50
Wielowierszowy ciąg tekstowy	51
Zastępowanie ciągu tekstowego	53
Szablony wraz z tagami	54
Podsumowanie	58

3

FUNKCJE 59

Funkcje wraz z wartościami parametrów domyślnych	59
Symulowanie wartości parametrów domyślnych w ECMAScript 5	60
Wartości parametrów domyślnych w ECMAScript 6	61
Jaki wpływ na argumenty <code>Object</code> mają wartości parametrów domyślnych?	62
Wyrażenia parametru domyślnego	64
Parametr domyślny i koncepcja TDZ	66
Praca z nienazwanymi parametrami	68
Nienazwane parametry w ECMAScript 5	68
Parametry resztowe	69
Zwiększone możliwości konstruktora <code>Function</code>	72
Operator rozszczepiania	72
Właściwość <code>name</code>	74
Wybór odpowiedniej nazwy	74
Przypadki specjalne dotyczące właściwości <code>name</code>	75
Wyjaśnienie podwójnego przeznaczenia funkcji	76
Ustalenie, jak funkcja była wywoływana w ECMAScript 5	77
Metawłaściwość <code>new.target</code>	78
Funkcje na poziomie bloku	79
Ustalenie, kiedy należy używać funkcji na poziomie bloku	80
Funkcje na poziomie bloku w trybie nieściśłym	81

4

Funkcja strzałki	81
Składnia funkcji strzałki	82
Utworzenie natychmiast wykonywanego wyrażenia funkcji	84
Brak wiązania this	85
Funkcja strzałki i tablica	88
Brak wiązania arguments	88
Identyfikacja funkcji strzałki	88
Optymalizacja wywołania ogonowego	89
Co nowego w wywołaniu ogonowym w specyfikacji ECMAScript 6?	90
Jak określić optymalizację wywołania ogonowego?	92
Podsumowanie	93

4

ROZBUDOWANA FUNKCJONALNOŚĆ OBIEKTU 95

Kategorie obiektu	95
Rozszerzenia składni literału obiektu	96
Skrót inicjalizacji właściwości	96
Zwięzłe metody	97
Generowane nazwy właściwości	98
Nowe metody	100
Metoda Object.is()	100
Metoda Object.assign()	101
Powielenie właściwości literału obiektu	103
Kolejność właściwości typu wyliczeniowego	104
Usprawnienia dla prototypów	105
Zmiana prototypu obiektu	105
Łatwy dostęp do prototypu za pomocą odwołania super	107
Formalna definicja metody	110
Podsumowanie	111

5

DESTRUKTURYZACJA W CELU ŁATWIEJSZEGO

DOSTĘPU DO DANYCH 113

Dlaczego destruktywizacja może być użyteczna?	113
Destruktywizacja obiektu	114
Destruktywizacja przypisania	115
Wartości domyślne	116
Przypisanie do różnych nazw zmiennych lokalnych	117
Destruktywizacja zagnieżdżonego obiektu	118
Destruktywizacja tablicy	120
Destruktywizacja przypisania	121
Wartości domyślne	123
Destruktywizacja zagnieżdżonej tablicy	123
Elementy resztowe	123

Destrukturyzacja mieszana	125
Destrukturyzacja parametrów	125
Destrukturyzowane parametry są wymagane	127
Wartości domyślne dla destrukturyzowanych parametrów	128
Podsumowanie	128

6

SYMBOLE I ICH WŁAŚCIWOŚCI 131

Utworzenie symbolu	132
Użycie symbolu	133
Współdzielenie symboli	134
Koercja symbolu	135
Pobieranie właściwości symbolu	136
Udostępnienie wewnętrznych operacji za pomocą powszechnie znanych symboli	137
Metoda Symbol.hasInstance	138
Właściwość Symbol.isConcatSpreadable	140
Właściwości Symbol.match, Symbol.replace, Symbol.search i Symbol.split	142
Metoda Symbol.toPrimitive	144
Właściwość Symbol.toStringTag	146
Właściwość Symbol.unscopables	150
Podsumowanie	151

7

ZBIORY I MAPY 153

Zbiory i mapy w ECMAScript 5	154
Problemy związane z obciążeniami	155
Zbiory w specyfikacji ECMAScript 6	156
Utworzenie zbioru i dodanie do niego elementów	156
Usunięcie elementu ze zbioru	158
Metoda forEach() dla zbioru	159
Konwersja zbioru na postać tablicy	161
Słaby zbiór	162
Mapy w specyfikacji ECMAScript 6	164
Metody mapy	165
Inicjalizacja mapy	166
Metoda forEach() dla mapy	167
Słabe mapy	168
Podsumowanie	172

8

ITERATORY I GENERATORY 175

Problem związany z pętlą	175
Czym jest iterator?	176
Czym jest generator?	177
Wyrażenie funkcji generatora	179
Metody obiektu generatora	180

Pętla for-of i elementy poddające się iteracji	181
Uzyskanie dostępu do domyślnego iteratora	182
Utworzenie obiektu poddającego się iteracji	183
Wbudowane iteratory	184
Iteratory kolekcji	184
Iteratory ciągu tekstowego	189
Iteratory NodeList	190
Operator rozszczepiania i niebędące tablicami elementy poddające się iteracji	191
Zaawansowana funkcjonalność iteratorów	192
Przekazanie argumentów do iteratora	192
Zgłaszanie błędów w iteratorze	194
Polecenie return w generatorze	196
Delegowanie generatora	197
Asynchroniczne wykonywanie zadania	199
Wykonywanie prostych zadań	200
Wykonanie zadania wraz z danymi	201
Asynchroniczne wykonywanie zadań	202
Podsumowanie	205

9

WPROWADZENIE DO KLAS JAVASCRIPT 207

Przypominające klasy struktury w ECMAScript 5	208
Deklaracja klasy	208
Podstawowa deklaracja klasy	208
Dlaczego należy używać składni klasy?	210
Wyrażenia klasy	212
Podstawowe wyrażenie klasy	212
Wyrażenia nazwanych klas	213
Klasa jako typ pierwszoklasowy	214
Właściwości akcesora	216
Generowane nazwy elementów składowych	217
Metody generatora	218
Statyczne elementy składowe	220
Dziedziczenie z użyciem klas pochodnych	221
Przesłanianie metod klasy	224
Dziedziczone statyczne elementy składowe	224
Klasy pochodne na podstawie wyrażeń	225
Dziedziczenie po wbudowanych klasach	228
Właściwość Symbol.species	229
Użycie właściwości new.target w konstruktorze klasy	232
Podsumowanie	234

10

USPRAWNIONE MOŻLIWOŚCI TABLICY 237

Tworzenie tablicy	237
Metoda <code>Array.of()</code>	238
Metoda <code>Array.from()</code>	239
Nowe metody we wszystkich tablicach	243
Metody <code>find()</code> i <code>findIndex()</code>	243
Metoda <code>fill()</code>	244
Metoda <code>copyWithin()</code>	245
Typowane tablice	246
Liczbowe typy danych	246
Bufor tablicy	247
Przeprowadzanie operacji na buforze tablicy za pomocą widoku	248
Podobieństwa między tablicami typowanymi i zwykłymi	255
Metody używane w obu typach tablic	256
Te same iteratory	256
Metody <code>of()</code> i <code>from()</code>	257
Różnice między tablicami typowaną i zwykłą	257
Różnice behawioralne	258
Brakujące metody	259
Metody dodatkowe	259
Podsumowanie	260

11

OBJETNICE I PROGRAMOWANIE ASYNCHRONICZNE 263

Kontekst programowania asynchronicznego	264
Model zdarzeń	264
Wzorzec wywołania zwrotnego	265
Podstawy obietnic	267
Cykl życiowy obietnicy	268
Tworzenie nierozstrzygniętej obietnicy	270
Utworzenie spełnionej obietnicy	273
Błędy funkcji <code>executor</code>	275
Globalna procedura obsługi odrzucenia obietnicy	276
Obsługa odrzucenia obietnicy w <code>Node.js</code>	277
Obsługa odrzucenia obietnicy w przeglądarce WWW	279
Łączenie obietnic	281
Przechwytywanie błędów	282
Wartość zwrotna w łańcuchu obietnic	283
Zwrot obietnicy przez łańcuch obietnic	284
Udzielanie odpowiedzi wielu obietnicom	287
Metoda <code>Promise.all()</code>	287
Metoda <code>Promise.race()</code>	288

Dziedziczenie po obietnicach	289
Asynchroniczne wykonywanie zadania za pomocą obietnicy	291
Podsumowanie	295

12

PROXY I API REFLEKSJI	297
Problem tablicy	298
Wprowadzenie proxy i refleksji	298
Utworzenie prostego proxy	300
Weryfikacja właściwości za pomocą pułapki set	300
Weryfikacja kształtu obiektu za pomocą pułapki get	302
Ukrycie istnienia właściwości za pomocą pułapki has	304
Uniemożliwienie usunięcia właściwości za pomocą pułapki deleteProperty	305
Pułapki prototypu proxy	307
Jak działają pułapki prototypu proxy?	308
Dlaczego mamy dwa zbiory metod?	309
Pułapki związane z rozbudową obiektu	311
Dwa proste przykłady	311
Powielone metody związane z rozbudową obiektów	312
Pułapki deskryptora właściwości	313
Blokowanie Object.defineProperty()	314
Ograniczenia deskryptora obiektu	315
Powielone metody deskryptora	316
Pułapka ownKeys	318
Funkcje proxy używane podczas konstruowania i stosowania pułapek	319
Weryfikacja parametrów funkcji	320
Wywołanie konstruktora bez operatora new	322
Nadpisanie konstruktora abstrakcyjnej klasy bazowej	323
Możliwy do wywołania konstruktor klasy	325
Proxy możliwe do odwołania	326
Rozwiązanie problemu tablicy	327
Wykrywanie indeksu tablicy	328
Zwiększenie wielkości po dodaniu nowego elementu	328
Usuwanie elementów po zmniejszeniu wartości właściwości length	330
Implementacja klasy MyArray	332
Użycie proxy jako prototypu	334
Użycie pułapki get w prototypie	335
Użycie pułapki set w prototypie	336
Użycie pułapki has w prototypie	337
Proxy jako prototyp w klasie	338
Podsumowanie	341

I 3

HERMETYZACJA KODU ZA POMOCĄ MODUŁÓW 343

Co to jest moduł?	344
Podstawowe operacje eksportu	344
Podstawowe operacje importu	345
Import pojedynczego wiązania	346
Import wielu wiązań	346
Import całego modułu	347
Drobne dziwactwo zaimportowanych wiązań	348
Zmiana nazwy elementu podczas eksportu i importu	349
Wartość domyślna w module	350
Eksport wartości domyślnej	350
Import wartości domyślnej	351
Ponowny eksport wiązania	352
Import bez wiązań	353
Wczytywanie modułu	354
Użycie modułu w przeglądarce WWW	354
Określanie specyfikatora modułu w przeglądarce WWW	359
Podsumowanie	360

A

MNIEJSZE ZMIANY W ECMASCRIPT 6 361

Praca z liczbami całkowitymi	361
Identyfikacja liczby całkowitej	362
Bezpieczne liczby całkowite	362
Nowe metody obiektu Math	363
Identyfikatory Unicode	364
Formalizacja właściwości __proto__	365

B

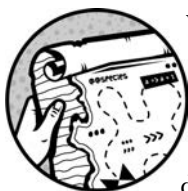
POZNAJEMY ECMASCRIPT 7 (2016) 369

Operator wykładniczy	370
Kolejność operacji	370
Ograniczenie operandu	370
Metoda Array.prototype.includes()	371
Jak używać metody Array.prototype.includes()?	372
Porównywanie wartości	372
Zmiana w trybie ścisłym o zasięgu funkcji	373

SKOROWIDZ 375

9

Wprowadzenie do klas JavaScript



W PRZECIWIENSTWIE DO WIĘKSZOŚCI FORMALNYCH JĘZYKÓW PROGRAMOWANIA ZORIENTOWANEGO OBIEKTOWO, JAVASCRIPT NIE OBSŁUGUJE KLAS I ICH DZIEDZICZENIA JAKO PODSTAWOWEGO SPOSOBU DEFINIOWANIA PODOBNYCH I POWIĄZANYCH ZE SOBĄ OBIEKTÓW W CHWILI ICH TWORZENIA. Takie podejście wprowadziło sporo zamieszania wśród programistów. Od specyfikacji w wersji sprzed ECMAScript 1 do ECMAScript 5 wiele bibliotek dostarczało narzędzia, dzięki którym mogło się wydawać, że JavaScript obsługuje klasy. Wprawdzie pewna część programistów była mocno przekonana, że język nie potrzebuje klas, ale ilość utworzonych bibliotek przeznaczonych do obsługi klas spowodowała dołączenie klas w specyfikacji ECMAScript 6.

Podczas poznawania klas w specyfikacji ECMAScript 6 pomocne będzie zrozumienie sposobu działania mechanizmu używanego przez klasy. Dlatego też na początek pokażę, jak programiści uzyskiwali w ECMAScript 5 zachowanie przypominające klasy. Jednak — jak się przekonasz — klasy wprowadzone w specyfikacji ECMAScript 6 nie przypominają klas znanych z innych języków programowania. Ich unikatowość wynika z dynamicznej natury języka JavaScript.

Przypominające klasy struktury w ECMAScript 5

Jak wcześniej wspomniałem, w specyfikacji ECMAScript 5 i wcześniejszych język JavaScript nie oferował klas. Najbliższym odpowiednikiem klasy było utworzenie konstruktora, a następnie przypisanie metod prototypowi konstruktora. Tego rodzaju podejście było zwykle określane mianem utworzenia własnego typu. Spójrz na poniższy fragment kodu.

```
function PersonType(name) {  
    this.name = name;  
}  
  
PersonType.prototype.sayName = function() {  
    console.log(this.name);  
};  
  
var person = new PersonType("Nicholas");  
person.sayName();    // Dane wyjściowe to "Nicholas".  
  
console.log(person instanceof PersonType);    // Wartość true.  
console.log(person instanceof Object);        // Wartość true.
```

W powyższym fragmencie kodu `PersonType` to funkcja konstruktora tworząca pojedynczą właściwość o nazwie `name`. Metoda `sayName()` jest przypisana prototypowi, aby ta sama funkcja mogła być współdzielona przez wszystkie egzemplarze obiektu `PersonType`. Kolejnym krokiem jest utworzenie nowego egzemplarza `PersonType` za pomocą operatora `new`. Otrzymany w wyniku obiekt `person` jest uznawany za egzemplarz `PersonType` i `Object` poprzez dziedziczenie prototypowe.

Tego rodzaju podstawowy wzorec jest stosowany w wielu bibliotekach JavaScript próbujących zapewnić obsługę klas w języku. To miejsce jest początkiem klas w specyfikacji ECMAScript 6.

Deklaracja klasy

Najprostszą postacią klasy w ECMAScript 6 jest deklaracja klasy, która przedstawia się podobnie do klas używanych w innych językach programowania.

Podstawowa deklaracja klasy

Deklaracja klasy rozpoczyna się od słowa kluczowego `class`, po którym znajduje się nazwa klasy. Pozostała część składni przypomina zwięzłe metody w literale obiektu, choć między elementami klasy nie są wymagane przecinki. Poniżej przedstawiłem przykład prostej deklaracji klasy.

```

class PersonClass {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName(); // Dane wyjściowe to "Nicholas".

console.log(person instanceof PersonClass); // Wartość true.
console.log(person instanceof Object); // Wartość true.

console.log(typeof PersonClass); // "function".
console.log(typeof PersonClass.prototype.sayName); // "function".

```

Deklaracja klasy dla `PersonClass` zachowuje się podobnie jak `PersonType` w poprzednim przykładzie. Jednak zamiast zdefiniować funkcję jako konstruktora, deklaracja klasy pozwala na zdefiniowanie konstruktora bezpośrednio w klasie za pomocą metody specjalnej o nazwie `constructor`. Ponieważ metody klasy używają zwięzłej składni, więc nie ma potrzeby stosowania słowa kluczowego `function`. Nazwy wszystkich pozostałych metod nie mają specjalnego znaczenia, możesz dodać dowolną liczbę metod.

Własne właściwości to właściwości egzemplarza, a nie prototypu, i dlatego mogą być tworzone tylko wewnątrz metody lub konstruktora klasy. W omawianym przykładzie `name` jest własną właściwością. Zalecam, aby w funkcji konstruktora utworzyć wszystkie możliwe własne właściwości. W ten sposób mamy w klasie pojedyncze miejsce odpowiedzialne za te wszystkie właściwości.

Interesujący może być fakt, że deklaracje klasy to po prostu syntaktyczny dodatek dla istniejących deklaracji typów niestandardowych. Deklaracja `Person` ↪ `Class` tworzy funkcję posiadającą zachowanie metody `constructor` i dlatego też wartością zwrótną wywołania `typeof PersonClass` jest `"function"`. Metoda `sayName()` w omawianym przykładzie również kończy jako metoda w `PersonClass`. ↪ `prototype`, podobnie jak w przypadku związku między `sayName()` i `PersonType`. ↪ `prototype` w poprzednich przykładach. Te podobieństwa pozwalają na łączenie niestandardowych typów i klas bez większych obaw o to, które z nich są używane.

UWAGA Prototypy klasy, takie jak `PersonClass.prototype` w poprzednim przykładzie, przeznaczone są tylko do odczytu. Oznacza to brak możliwości przypisania nowej wartości prototypowi, co można zrobić w przypadku funkcji.

Dlaczego należy używać składni klasy?

Pomimo pewnych podobieństw między klasami i typami niestandardowymi, należy pamiętać o istnieniu ważnych różnic między nimi.

- W przeciwieństwie do deklaracji funkcji, deklaracje klas nie podlegają hoistingowi. Deklaracja klasy działa w charakterze deklaracji `let`, więc istnieje w tymczasowo martwej strefie aż do chwili, gdy wykonywanie programu dotrze do tej deklaracji.
- Cały kod znajdujący się w deklaracji klasy jest automatycznie uruchamiany w trybie ścisłym. Nie ma żadnej możliwości rezygnacji z trybu ścisłego wewnątrz klasy.
- Żadna z metod nie poddaje się wyliczeniu. To istotna zmiana w stosunku do typu niestandardowego, w którym trzeba użyć wywołania `Object.defineProperty()`, aby metoda nie poddawała się wyliczeniu.
- Żadna metoda nie ma wewnętrznej metody `[[Construct]]` i zgłosi błąd, jeśli spróbujesz wywoływać ją wraz ze słowem kluczowym `new`.
- Wywołanie konstruktora klasy bez słowa kluczowego `new` powoduje zgłoszenie błędu.
- Próba nadpisania nazwy klasy metodą klasy powoduje zgłoszenie błędu.

Mając na uwadze te wszystkie różnice, możemy zobaczyć, że przedstawiona w poprzednim przykładzie deklaracja `PersonClass` jest bezpośrednim odpowiednikiem poniższego kodu, w którym nie użyliśmy składni klasy.

// Bezpośredni odpowiednik PersonClass.

```
let PersonType2 = (function() {

    "use strict";

    const PersonType2 = function(name) {

        // Upewniamy się, że funkcja została wywołana wraz ze słowem kluczowym new.
        if (typeof new.target === "undefined") {
            throw new Error("Konstruktor musi być wywołany wraz ze słowem
            ↪kluczowym new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonType2.prototype, "sayName", {
        value: function() {

            // Upewniamy się, że metoda nie została wywołana wraz ze słowem
            // kluczowym new.
            if (typeof new.target !== "undefined") {
```

```

        throw new Error("Metoda nie może być wywołana wraz ze
        ↪ słowem kluczowym new.");
    }

    console.log(this.name);
  },
  enumerable: false,
  writable: true,
  configurable: true
});

return PersonType2;
})();

```

Przed wszystkim zwróć uwagę na istnienie dwóch deklaracji `PersonType2`. Pierwsza to deklaracja `let` w zewnętrznym zasięgu, natomiast druga to deklaracja `const` w natychmiast wykonywanym wyrażeniu funkcji. Teraz już wiesz, dlaczego metody klasy nie mogą nadpisywać nazwy klasy, choć może to zrobić kod zdefiniowany poza klasą. Funkcja konstruktora sprawdza `new.target`, aby mieć pewność wywołania wraz ze słowem kluczowym `new`. W przeciwnym razie nastąpi zgłoszenie błędu. Następnie metoda `sayName()` zostaje zdefiniowana jako niepoddająca się wyliczeniu i sprawdza `new.target`, aby mieć pewność, że nie była wywołana wraz z `new`. Ostatnim krokiem jest zwrot funkcji konstruktora.

W omówionym przykładzie pokazałem, że można wykorzystać możliwości klas bez użycia nowej składni, ale ta właśnie składnia znacznie upraszcza funkcjonalność.

NAZWY STAŁYCH KLASY

Nazwa klasy to jedyna stała w klasie. Oznacza to, że można nadpisać nazwę klasy na zewnątrz niej, ale nie w metodzie klasy. Spójrz na poniższy fragment kodu.

```

class Foo {
  constructor() {
    Foo = "bar"; // Zgłoszenie błędu podczas wykonywania kodu...
  }
}

// Takie podejście jest dozwolone, ale po deklaracji klasy.
Foo = "baz";

```

W powyższym fragmencie kodu `Foo` wewnątrz konstruktora klasy to wiązanie oddzielne od `Foo` na zewnątrz klasy. Wewnętrzne `Foo` zostało zdefiniowane jako stała, której nie można nadpisać. Kiedy konstruktor spróbuje nadpisać to `Foo` dowolną wartością, zostanie zgłoszony błąd. Z kolei zewnętrzne `Foo` zostało zdefiniowane za pomocą deklaracji `let` i dlatego w każdym momencie można nadpisać jego wartość.

Wyrażenia klasy

Klasy i funkcje są podobne pod tym względem, że mają dwie formy — deklaracje i wyrażenia. Deklaracja funkcji i klasy rozpoczyna się od właściwego słowa kluczowego (odpowiednio — `function` i `class`), po którym znajduje się identyfikator. Funkcje mają formę wyrażenia niewymagającą podawania identyfikatora po słowie kluczowym `function`. Podobnie klasy mają formę wyrażenia niewymagającą podawania identyfikatora po słowie kluczowym `class`. Te **wyrażenia klasy** zostały zaprojektowane do użycia w deklaracjach zmiennych lub przekazywania funkcjom jako argumenty.

Podstawowe wyrażenie klasy

Poniżej przedstawiłem odpowiednik wyrażenia klasy dla wcześniejszych przykładów `PersonClass`. Następnie znajduje się pewien kod, w którym użyto tych wyrażen klasy.

```
let PersonClass = class {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
    }
};

let person = new PersonClass("Nicholas");
person.sayName();    // Dane wyjściowe to "Nicholas".

console.log(person instanceof PersonClass);    // Wartość true.
console.log(person instanceof Object);         // Wartość true.

console.log(typeof PersonClass);               // "function".
console.log(typeof PersonClass.prototype.sayName); // "function".
```

Jak możesz zobaczyć w powyższym fragmencie kodu, wyrażenie klasy nie wymaga identyfikatora po słowie kluczowym `class`. Kiedy pominiemy składnię, wyrażenia klasy są pod względem funkcjonalnym odpowiednikami deklaracji klas. W wyrażeniach klas anonimowych, jak w powyższym przykładzie, `PersonClass`. `name` to pusty ciąg tekstowy. Kiedy używasz deklaracji klasy, wówczas wartością `PersonClass.name` będzie ciąg tekstowy `"PersonClass"`.

To, czy korzystasz z deklaracji klasy, czy z wyrażenia klasy, to przede wszystkim kwestia stylu. W przeciwieństwie do deklaracji i wyrażen funkcji, deklaracje

i wyrażenia klasy nie są poddawane hoistingowi, więc wybór nie ma większego wpływu na zachowanie kodu podczas jego wykonywania. Jedyna istotna różnica polega na tym, że wyrażenia klas anonimowych mają właściwość `name` o wartości w postaci pustego ciągu tekstowego, natomiast deklaracje klas zawsze mają właściwość `name` o wartości odpowiadającej nazwie klasy (gdy np. używasz deklaracji klasy, wartością `PersonClass.name` jest `"PersonClass"`).

Wyrażenia nazwanych klas

W poprzednim przykładzie używaliśmy wyrażen klas anonimowych, ale podobnie jak w wyrażeniach funkcji, można również stosować wyrażenia nazwanych klas. W tym celu po słowie kluczowym `class` należy umieścić identyfikator, np. tak jak pokazałem poniżej.

```
let PersonClass = class PersonClass2 {  
  
    // Odpowiednik konstruktora PersonType.  
    constructor(name) {  
        this.name = name;  
    }  
  
    // Odpowiednik PersonType.prototype.sayName.  
    sayName() {  
        console.log(this.name);  
    }  
};  
  
console.log(typeof PersonClass);           // "function".  
console.log(typeof PersonClass2);         // "undefined".
```

W powyższym przykładzie wyrażenie klasy nosi nazwę `PersonClass2`. Ten identyfikator istnieje tylko w definicji klasy, więc może być używany wewnątrz metod klasy (np. w metodzie `sayName()`). Poza klasą wywołanie `typeof PersonClass2` zwróci wartość `"undefined"`, ponieważ nie istnieje tutaj żadne wiązanie `Person` → `Class2`. Aby zrozumieć, dlaczego tak się dzieje, spójrz na odpowiadającą poprzedniemu przykładowi deklarację, która nie używa klas.

```
// Bezpośredni odpowiednik wyrażenia nazwanej klasy PersonClass.  
let PersonClass = (function() {  
  
    "use strict";  
  
    const PersonClass2 = function(name) {  
  
        // Upewniamy się, że funkcja została wywołana wraz ze słowem kluczowym new.  
        if (typeof new.target === "undefined") {
```

```

        throw new Error("Konstruktor musi być wywołany wraz ze słowem
        ↪kluczowym new.");
    }

    this.name = name;
}

Object.defineProperty(PersonClass2.prototype, "sayName", {
    value: function() {

        // Upewniamy się, że metoda nie została wywołana wraz ze słowem
        // kluczowym new.
        if (typeof new.target !== "undefined") {
            throw new Error("Metoda nie może być wywołana wraz
            ↪ze słowem kluczowym new.");
        }

        console.log(this.name);
    },
    enumerable: false,
    writable: true,
    configurable: true
});

return PersonClass2;
})();

```

Utworzenie wyrażenia nazwanej klasy nieco zmienia to, co się dzieje w silniku JavaScript. W przypadku deklaracji klasy zewnętrzne wiązanie (zdefiniowane za pomocą `let`) ma taką samą nazwę jak wiązanie wewnętrzne (zdefiniowane za pomocą `const`). W wyrażeniu nazwanej klasy zastosowano nazwę z definicji `const` i dlatego wyrażenie `PersonClass2` jest zdefiniowane do użycia tylko wewnątrz klasy.

Wprawdzie wyrażenia nazwanych klas zachowują się nieco inaczej niż wyrażenia nazwanych funkcji, ale między nimi występują także pewne podobieństwa. Oba rodzaje wyrażeń mogą być używane jako wartości, co otwiera wiele możliwości, którymi zajmę się w kolejnym podrozdziale.

Klasa jako typ pierwszoklasowy

W programowaniu **typ pierwszoklasowy** (ang. *first-class citizen*) to wartość, która może być przekazana funkcji, zwrócona przez funkcję i przypisana zmiennej. Funkcje JavaScript są typami pierwszoklasowymi (nazywane są również **funkcjami pierwszoklasowymi**), które jednocześnie stanowią o unikatowości języka JavaScript.

Specyfikacja ECMAScript 6 kontynuuje tę tradycję przez uznanie klasy także za typ pierwszoklasowy, co pozwala na ich użycie na wiele różnych sposobów. Przykładowo klasę można przekazać funkcji jako argument, tak jak pokażę poniżej.

```
function createObject(classDef) {  
    return new classDef();  
}  
  
let obj = createObject(class {  
    sayHi() {  
        console.log("Cześć!");  
    }  
});  
  
obj.sayHi();           // "Cześć!"
```

W powyższym fragmencie kodu funkcja `createObject()` jest wywoływana wraz z argumentem w postaci wyrażenia klasy anonimowej, tworzy egzemplarz tej klasy za pomocą słowa kluczowego `new` i zwraca ten egzemplarz. Następnie zmienna `obj` przechowuje zwrócony egzemplarz.

Innym przykładem użycia wyrażenia klasy jest utworzenie egzemplarza typu singleton za pomocą natychmiast wywoływanego konstruktora klasy. W tym celu trzeba użyć słowa kluczowego `new` wraz z wyrażeniem klasy i umieścić na końcu nawias. Tego rodzaju rozwiązanie przedstawiłem w poniższym fragmencie kodu.

```
let person = new class {  
    constructor(name) {  
        this.name = name;  
    }  
    sayName() {  
        console.log(this.name);  
    }  
}("Nicholas");  
  
person.sayName();    // "Nicholas"
```

W omawianym przykładzie wyrażenie klasy anonimowej zostaje utworzone, a później natychmiast wykonane. Wzorec ten pozwala na skorzystanie ze składni klasy do tworzenia egzemplarzy typu singleton bez pozostawienia dostępnego odwołania do klasy w celu analizy. Nawias na końcu wyrażenia klasy wskazuje na wywołanie funkcji, a ponadto pozwala na przekazanie argumentu.

Przykłady przedstawione dotąd w rozdziale były skoncentrowane na klasach wraz z metodami. Jednak istnieje również możliwość utworzenia właściwości akcesora w klasach za pomocą składni podobnej do stosowanej w literałach obiektów.

Właściwości akcesora

Wprawdzie powinienś tworzyć własne właściwości w konstruktorze klasy, ale klasa pozwala też na zdefiniowanie właściwości akcesora w prototypie. W celu utworzenia metody getter użyj słowa kluczowego `get`, spacji i identyfikatora. Natomiast w celu utworzenia metody setter zastosuj takie samo podejście, ale użyj słowa kluczowego `set`. Spójrz na poniższy fragment kodu.

```
class CustomHTMLElement {
  constructor(element) {
    this.element = element;
  }

  get html() {
    return this.element.innerHTML;
  }

  set html(value) {
    this.element.innerHTML = value;
  }
}

var descriptor =
Object.getOwnPropertyDescriptor(CustomHTMLElement.prototype, "html");
console.log("get" in descriptor);           // Wartość true.
console.log("set" in descriptor);           // Wartość true.
console.log(descriptor.enumerable);          // Wartość false.
```

W omawianym przykładzie klasa `CustomHTMLElement` została użyta w charakterze opakowania dla istniejącego elementu modelu DOM. Ma metody getter i setter dla właściwości `html` delegującej do metody `innerHTML` elementu. Ta właściwość akcesora została utworzona w `CustomHTMLElement.prototype` i podobnie jak każda inna metoda, także i ona nie poddaje się wyliczeniu. Poniżej przedstawiłem niezawierający klasy odpowiednik omówionego rozwiązania.

```
// Bezpośredni odpowiednik poprzedniego przykładu.
let CustomHTMLElement = (function() {

  "use strict";

  const CustomHTMLElement = function(element) {
```

```

// Upewniamy się, że funkcja została wywołana wraz ze słowem kluczowym new.
if (typeof new.target === "undefined") {
    throw new Error("Konstruktor musi być wywołany wraz ze słowem
    ↪kluczowym new.");
}

this.element = element;
}

Object.defineProperty(CustomHTMLElement.prototype, "html", {
    enumerable: false,
    configurable: true,
    get: function() {
        return this.element.innerHTML;
    },
    set: function(value) {
        this.element.innerHTML = value;
    }
});

return CustomHTMLElement;
})();

```

Przykład ten, podobnie jak poprzednie, także pokazuje, jak można uniknąć konieczności utworzenia sporej ilości kodu przy użyciu klasy, a nie odpowiednika niezawierającego klasy. Definicja samej właściwości akcesora `html` zabiera prawie połowę wielkości odpowiednika deklaracji klasy.

Generowane nazwy elementów składowych

Podobieństw między literałami obiektu i klasami istnieje znacznie więcej. Metody klasy i właściwości akcesora mogą mieć wygenerowane nazwy. Zamiast korzystać z identyfikatora, należy użyć nawiasu kwadratowego zawierającego wyrażenie. Takie podejście odpowiada składni używanej do generowania nazw literałów obiektu. Spójrz na poniższy fragment kodu.

```

let methodName = "sayName";

class PersonClass {

    constructor(name) {
        this.name = name;
    }
    [methodName]() {
        console.log(this.name);
    }
}

```

```
};  
  
let me = new PersonClass("Nicholas");  
me.sayName();           // "Nicholas".
```

Powyższa wersja `PersonClass` używa zmiennej w celu przypisania nazwy metody wewnątrz jej definicji. Ciąg tekstowy `"sayName"` zostaje przypisany zmiennej `methodName`, która następnie będzie użyta do zadeklarowania metody. Później dostęp do metody `sayName()` odbywa się bezpośrednio.

Właściwości akcesora mogą dokładnie tak samo używać wygenerowanych nazw. Spójrz na poniższy fragment kodu.

```
let propertyName = "html";  
  
class CustomHTMLElement {  
    constructor(element) {  
        this.element = element;  
    }  
  
    get [propertyName]() {  
        return this.element.innerHTML;  
    }  
  
    set [propertyName](value) {  
        this.element.innerHTML = value;  
    }  
}
```

W omawianym przykładzie metody `getter` i `setter` dla `html` zostały zdefiniowane za pomocą zmiennej `propertyName`. Uzyskanie dostępu do właściwości przy użyciu składni `.html` wpływa jedynie na definicję.

Poznałeś wiele podobieństw między klasami i literalami obiektów, m.in. metody, właściwości akcesora i generowane nazwy. Istnieje jeszcze jedno podobieństwo, którym chciałbym się zająć, czyli generatory.

Metody generatora

Przypomnij sobie z rozdziału 8. o możliwości zdefiniowania generatora w literale obiektu przez poprzedzenie gwiazdką nazwy metody. Ta sama składnia sprawdza się również w przypadku klas i pozwala, aby dowolna metoda stała się generatorem. Poniżej przedstawiłem przykład tego rodzaju rozwiązania.

```
class MyClass {  
  
    *createIterator() {
```

```

        yield 1;
        yield 2;
        yield 3;
    }
}

let instance = new MyClass();
let iterator = instance.createIterator();

```

W powyższym fragmencie kodu powstała klasa o nazwie `MyClass` wraz z metodą generatora `createIterator()`. Metoda zwraca iterator, którego wartości są na stałe zdefiniowane w generatorze. Metody generatorów okazują się użyteczne, gdy mamy obiekt przedstawiający kolekcję wartości, przez które chcemy łatwo przeprowadzić iterację. Dla tablic, zbiorów i map istnieje sporo metod generatorów, co pozwala programistom pracować na wiele różnych sposobów z obiektami wymienionych typów.

Wprawdzie metody generatorów są użyteczne, ale zdefiniowanie domyślnego iteratora dla klasy będzie znacznie bardziej przydatne, jeśli klasa ta przedstawia kolekcję wartości. Aby zdefiniować domyślny iterator dla klasy, można użyć `Symbol.iterator` w celu określenia metody generatora. Spójrz na poniższy fragment kodu.

```

class Collection {
    constructor() {
        this.items = [];
    }

    *[Symbol.iterator]() {
        yield *this.items.values();
    }
}

var collection = new Collection();
collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}

// Dane wyjściowe:
// 1
// 2
// 3

```

W przykładzie użyto nazwy wygenerowanej dla generatora, który deleguje iterator `values()` do tablicy `this.items`. Każda klasa zarządzająca kolekcją wartości powinna zawierać domyślny iterator, ponieważ pewne operacje typowe dla

kolekcji wymagają jego dostępności. Teraz możesz już dowolnego egzemplarza `Collection` użyć bezpośrednio w pętli `for-of` lub wraz z operatorem rozszczepiania.

Dodanie metod i właściwości akcesora do prototypu klasy jest użyteczne, gdy chcesz je pokazać w egzemplarzach obiektu. Jeżeli z kolei potrzebujesz metod lub właściwości akcesora w klasie, wówczas musisz użyć statycznych elementów składowych.

Statyczne elementy składowe

Dodanie metod bezpośrednio w konstruktorze w celu symulacji statycznych elementów składowych to kolejny wzorec często stosowany w kodzie zgodnym ze specyfikacją ECMAScript 5 i wcześniejszymi. Spójrz na poniższy fragment kodu.

```
function PersonType(name) {
    this.name = name;
}

// Metoda statyczna.
PersonType.create = function(name) {
    return new PersonType(name);
};

// Metoda egzemplarza.
PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = PersonType.create("Nicholas");
```

W innych językach programowania metoda fabryki o nazwie `PersonType.create()` będzie uznawana za metodę statyczną, ponieważ jej dane nie zależą od egzemplarza typu `PersonType`. Klasa w specyfikacji ECMAScript 6 ułatwia tworzenie statycznych elementów składowych przez użycie formalnej adnotacji `static` przed nazwą metody lub właściwości akcesora. Poniżej przedstawiłem oparty na klasie odpowiednik poprzedniego przykładu.

```
class PersonClass {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
    }
}
```

```

    }

    // Odpowiednik PersonType.create.
    static create(name) {
        return new PersonClass(name);
    }
}

let person = PersonClass.create("Nicholas");

```

Definicja `PersonClass` ma pojedynczą metodę statyczną o nazwie `create()`. Składnia metody jest taka sama jak składnia użyta dla `sayName()`, z wyjątkiem słowa kluczowego `static`. Istnieje możliwość użycia słowa kluczowego `static` w definicji dowolnej metody lub właściwości akcesora w klasie. Jedyne ograniczenie wiąże się z brakiem możliwości użycia słowa kluczowego `static` w definicji metody `constructor`.

UWAGA *Statyczne elementy składowe są nieodstępne z poziomu egzemplarzy. Dlatego też dostęp do statycznych elementów składowych musi odbywać się bezpośrednio z poziomu klasy.*

Dziedziczenie z użyciem klas pochodnych

Przed wprowadzeniem specyfikacji ECMAScript 6 implementacja dziedziczenia wraz z typami niestandardowymi była dość obszernym procesem. Zastosowanie prawidłowego dziedziczenia wymagało wielu kroków. Spójrz na poniższy fragment kodu.

```

function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function Square(length) {
    Rectangle.call(this, length, length);
}

Square.prototype = Object.create(Rectangle.prototype, {
    constructor: {
        value: Square,
        enumerable: true,

```

```

        writable: true,
        configurable: true
    }
});

var square = new Square(3);

console.log(square.getArea());           // 9.
console.log(square instanceof Square);    // Wartość true.
console.log(square instanceof Rectangle); // Wartość true.

```

Obiekt `Square` dziedziczy po `Rectangle` i dlatego musi nadpisywać `Square.prototype` nowym obiektem utworzonym przez `Rectangle.prototype`, jak również wywołać metodę `Rectangle.call()`. Kroki te są często niezrozumiałe dla początkujących programistów JavaScript, a także stanowią źródło błędów dla doświadczonych programistów.

Klasy ułatwiają implementację dziedziczenia za pomocą znanego słowa kluczowego `extends` pozwalającego na wskazanie funkcji, po której dana klasa powinna dziedziczyć. Prototypy są automatycznie dostosowywane i można uzyskać dostęp do bazowego konstruktora klasy przez wywołanie metody `super()`. Poniżej przedstawiłem zgodny ze specyfikacją ECMAScript 6 odpowiednik poprzedniego przykładu.

```

class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {

        // Odpowiednik wywołania Rectangle.call(this, length, length).
        super(length, length);
    }
}

var square = new Square(3);

console.log(square.getArea());           // 9.
console.log(square instanceof Square);    // Wartość true.
console.log(square instanceof Rectangle); // Wartość true.

```

Tym razem klasa Square dziedziczy po Rectangle za pomocą słowa kluczowego `extends`. Konstruktor Square używa `super()` do wywołania konstruktora Rectangle wraz z podanymi argumentami. Warto zwrócić uwagę, że w przeciwieństwie do wersji kodu dla specyfikacji ECMAScript 5, identyfikator `Rectangle` jest używany jedynie w deklaracji klasy (po słowie kluczowym `extends`).

Klasy dziedziczące po innych klasach są określane mianem **klas pochodnych**. Klasa pochodna wymaga użycia wywołania `super()`, jeśli jest podawany konstruktor, w przeciwnym razie nastąpi zgłoszenie błędu. Jeżeli zdecydujesz się na nieużywanie konstruktora, metoda `super()` będzie wywoływana automatycznie po utworzeniu nowego egzemplarza klasy. Przykładowo dwie poniższe klasy są identyczne.

```
class Square extends Rectangle {  
    // Brak konstruktora.  
}  
  
// Powyższa definicja jest odpowiednikiem poniższej.  
  
class Square extends Rectangle {  
    constructor(...args) {  
        super(...args);  
    }  
}
```

Druga klasa w omawianym przykładzie pokazuje odpowiednik konstruktora domyślnego dla wszystkich klas pochodnych. Konstruktorowi klasy bazowej są po kolei przekazywane wszystkie argumenty. W omawianym przykładzie funkcjonalność nie jest całkiem poprawna, ponieważ konstruktor Square wymaga tylko jednego argumentu i dlatego najlepszym rozwiązaniem będzie ręczne zdefiniowanie konstruktora.

UWAGI DOTYCZĄCE UŻYCIA METODY `SUPER()`

Podczas stosowania metody `super()` pamiętaj o wymienionych poniżej kwestiach.

- ◆ Metody `super()` można używać jedynie w konstruktorze klasy pochodnej. Jeżeli spróbujesz ją wywołać w innej klasie (czyli nieużywającej słowa kluczowego `extends`) lub w funkcji, wówczas nastąpi zgłoszenie błędu.
- ◆ Metodę `super()` trzeba wywołać przed uzyskaniem dostępu do wiązania `this` w konstruktorze. Ponieważ metoda jest odpowiedzialna za inicjalizację `this`, więc próba uzyskania dostępu do `this` przed wywołaniem `super()` skutkuje zgłoszeniem błędu.
- ◆ Jedynym sposobem na uniknięcie wywołania metody `super()` jest zwrot obiektu przy użyciu konstruktora klasy.

Przesłanianie metod klasy

Metody w klasach pochodnych zawsze przesłaniają metody o takich samych nazwach zdefiniowane w klasie bazowej. Możesz np. dodać metodę `getAdd()` do `Square`, aby przeddefiniować tę funkcjonalność.

```
class Square extends Rectangle {
  constructor(length) {
    super(length, length);
  }

  // Nadpisanie i przesłonięcie metody Rectangle.prototype.getArea().
  getArea() {
    return this.length * this.length;
  }
}
```

Ponieważ metoda `getArea()` została zdefiniowana jako część egzemplarza `Square`, więc metoda o nazwie `Rectangle.prototype.getArea()` nie będzie dłużej wywoływana przez jakikolwiek egzemplarz `Square`. Oczywiście zawsze możesz się zdecydować na wywołanie metody w wersji zdefiniowanej w klasie bazowej. W omawianym przypadku oznacza to konieczność użycia wywołania `super.getArea()`, tak jak pokazałem poniżej.

```
class Square extends Rectangle {
  constructor(length) {
    super(length, length);
  }

  // Nadpisanie, przesłonięcie i wywołanie metody Rectangle.prototype.getArea().
  getArea() {
    return super.getArea();
  }
}
```

Użycie `super` w przedstawiony sposób działa dokładnie tak samo jak odwołanie `super` omówione w rozdziale 4. Wartość `this` będzie automatycznie prawidłowo ustawiona, aby można było wykonać proste wywołanie metody.

Dziedziczone statyczne elementy składowe

Jeżeli klasa bazowa ma statyczne elementy składowe, będą one dostępne również w klasie pochodnej. Dziedziczenie działa podobnie jak w innych językach programowania, ale jest to nowa koncepcja w JavaScriptcie. Spójrz na poniższy fragment kodu.

```
class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
  }

  getArea() {
    return this.length * this.width;
  }

  static create(length, width) {
    return new Rectangle(length, width);
  }
}

class Square extends Rectangle {
  constructor(length) {

    // Odpowiednik Rectangle.call(this, length, length).
    super(length, length);
  }
}

var rect = Square.create(3, 4);

console.log(rect instanceof Rectangle);    // Wartość true.
console.log(rect.getArea());              // 12.
console.log(rect instanceof Square);       // Wartość false.
```

W powyższym fragmencie kodu do klasy `Rectangle` została dodana nowa metoda statyczna o nazwie `create()`. Poprzez mechanizm dziedziczenia metoda ta stała się dostępna jako `Square.create()` i zachowuje się jak metoda `Rectangle.create()`.

Klasy pochodne na podstawie wyrażeń

Prawdopodobnie oferującym największe możliwości aspektem klas pochodnych w specyfikacji ECMAScript 6 jest możliwość utworzenia klasy pochodnej na podstawie wyrażenia. Słowa kluczowego `extends` można użyć wraz z dowolnym wyrażeniem, o ile będzie się ono sprowadzało do funkcji wraz z metodą `[[Construct]]` i prototypem. Spójrz na poniższy fragment kodu.

```
function Rectangle(length, width) {
  this.length = length;
  this.width = width;
}
```

```

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());           // 9.
console.log(x instanceof Rectangle); // Wartość true.

```

Konstruktor `Rectangle` to konstruktor zdefiniowany w stylu ECMAScript 5, natomiast `Square` to klasa. Ponieważ `Rectangle` zawiera metodę `[[Construct]]` i prototyp, więc klasa `Square` nadal może bezpośrednio po nim dziedziczyć.

Akceptacja dowolnego typu wyrażenia po słowie kluczowym `extends` oferuje użyteczne możliwości, takie jak dynamiczne ustalenie, co i po czym należy dziedziczyć. Spójrz na poniższy fragment kodu.

```

function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function getBase() {
    return Rectangle;
}

class Square extends getBase() {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());           // 9.
console.log(x instanceof Rectangle); // Wartość true.

```

Funkcja `getBase()` jest wywoływana bezpośrednio jako część deklaracji klasy. Zwraca egzemplarz `Rectangle` i dlatego pod względem funkcjonalności przykład ten jest odpowiednikiem poprzedniego. Ponieważ klasę bazową można

ustalić dynamicznie, więc stało się możliwe zastosowanie innych podejść podczas dziedziczenia. Można np. utworzyć domieszkę, tak jak pokazałem poniżej.

```
let SerializableMixin = {
  serialize() {
    return JSON.stringify(this);
  }
};

let AreaMixin = {
  getArea() {
    return this.length * this.width;
  }
};

function mixin(...mixins) {
  var base = function() {};
  Object.assign(base.prototype, ...mixins);
  return base;
}

class Square extends mixin(AreaMixin, SerializableMixin) {
  constructor(length) {
    super();
    this.length = length;
    this.width = length;
  }
}

var x = new Square(3);
console.log(x.getArea());           // 9.
console.log(x.serialize());         // '{"length":3,"width":3}'
```

W przykładzie użyłem domieszki zamiast klasycznego dziedziczenia. Funkcja `mixin()` pobiera dowolną liczbę argumentów przedstawiających obiekty domieszki. Tworzy funkcję o nazwie `base` i przypisuje prototypowi właściwości poszczególnych obiektów domieszki. Następnie funkcja zostaje zwrócona, aby możliwe było użycie słowa kluczowego `extends` wraz ze `Square`. Pamiętaj, że w konstruktorze wciąż konieczne będzie wywołanie `super()`, ponieważ nadal jest używane słowo kluczowe `extends`.

Exemplarz `Square` zawiera metody `getArea()` z `AreaMixin` oraz `serialize()` z `SerializableMixin`. Stało się to możliwe za sprawą dziedziczenia prototypowego. Funkcja `mixin()` dynamicznie wypełnia prototyp nową funkcją wraz z własnymi właściwościami poszczególnych domieszek. Warto pamiętać, że jeśli kilka domieszek ma tę samą właściwość, wówczas w nowym obiekcie pozostanie tylko ostatnia z dodanych.

UWAGA Wprawdzie po słowie kluczowym `extends` można użyć dowolnego wyrażenia, ale nie wszystkie spowodują powstanie prawidłowej klasy. Szczególnie użycie po `extends` wyrażenia zwracającego wartość `null` lub funkcję generatora (omówione w rozdziale 8.) spowoduje zgłoszenie błędu. W takim przypadku próba utworzenia nowego egzemplarza klasy spowoduje zgłoszenie błędu, ponieważ nie będzie metody `[[Construct]]` do wywołania.

Dziedziczenie po wbudowanych klasach

Niemal od samego początku istnienia tablic w języku JavaScript programiści chcieli mieć możliwość tworzenia własnych specjalnych typów tablic za pomocą dziedziczenia. W specyfikacji ECMAScript 5 i wcześniejszych nie było to możliwe. Próba zastosowania klasycznego dziedziczenia nie skutkowała powstaniem prawidłowo funkcjonującego kodu. Spójrz na poniższy fragment kodu.

```
// Zachowanie wbudowanej tablicy.
var colors = [];
colors[0] = "czerwony";
console.log(colors.length);           // 1.

colors.length = 0;
console.log(colors[0]);                // undefined.

// Próba dziedziczenia po tablicy w specyfikacji ES5.

function MyArray() {
    Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
    constructor: {
        value: MyArray,
        writable: true,
        configurable: true,
        enumerable: true
    }
});

var colors = new MyArray();
colors[0] = "czerwony";
console.log(colors.length);           // 0.

colors.length = 0;
console.log(colors[0]);                // "czerwony".
```

Wywołanie `console.log()` na końcu tego fragmentu kodu pokazuje, że użycie w JavaScriptcie klasycznej postaci dziedziczenia po tablicy powoduje nieoczekiwane zachowanie. Właściwość `length` i właściwości liczbowe w egzemplarzu

MyArray nie zachowują się w taki sam sposób jak we wbudowanej tablicy, ponieważ ta funkcjonalność nie jest obsługiwana przez `Array.apply()` lub za pomocą przypisania prototypu.

Jednym z celów klas w specyfikacji ECMAScript 6 było umożliwienie dziedziczenia po wszystkich wbudowanych typach danych. Aby osiągnąć ten cel, model dziedziczenia klas jest pod dwoma ważnymi względami nieco inny niż model dziedziczenia klasycznego znany ze specyfikacji ECMAScript 5 i wcześniejszych.

W przypadku klasycznego dziedziczenia w specyfikacji ECMAScript 5 wartość `this` jest najpierw tworzona przez typ pochodny (np. `MyArray`), a następnie mamy wywołanie konstruktora typu bazowego (np. metody `Array.apply()`). Oznacza to, że `this` na początku jest egzemplarzem `MyArray`, a następnie zostanie udekorowane dodatkowymi właściwościami typu `Array`.

Natomiast w opartym na klasach dziedziczeniu w specyfikacji ECMAScript 6 wartość `this` jest najpierw tworzona przez typ bazowy (tutaj `Array`), a dopiero później modyfikowana przez konstruktor klasy pochodnej (tutaj `MyArray`). Dlatego też `this` na początku ma całą funkcjonalność wbudowanego typu bazowego i prawidłowo otrzymuje wszelką związaną z nim funkcjonalność.

W poniższym przykładzie pokazałem w działaniu specjalną tablicę opartą na klasie.

```
class MyArray extends Array {  
    // Pusta definicja.  
}  
  
var colors = new MyArray();  
colors[0] = "czerwony";  
console.log(colors.length);           // 1.  
  
colors.length = 0;  
console.log(colors[0]);               // undefined.
```

Egzemplarz `MyArray` dziedziczy bezpośrednio po `Array` i dlatego działa podobnie jak tablica. Użycie właściwości liczbowych powoduje uaktualnienie właściwości `length`, a zmiana właściwości `length` pociąga za sobą uaktualnienie właściwości liczbowych. Oznacza to, że można poprawnie dziedziczyć po wbudowanym typie `Array`, aby utworzyć własne klasy pochodne tablicy, a także prawidłowo dziedziczyć po innych wbudowanych typach danych. Po dodaniu tej funkcjonalności w specyfikacji ECMAScript 6 i klasach pochodnych praktycznie usunięto ostatni przypadek specjalny dziedziczenia po wbudowanych typach danych. Mimo wszystko, nadal warto poznać wspomniany przypadek specjalny.

Właściwość `Symbol.species`

Wygodnym aspektem dziedziczenia po wbudowanych typach danych jest to, że każda metoda zwracająca egzemplarz wbudowanego typu danych będzie automatycznie zamiast niego zwracała egzemplarz klasy pochodnej. Dlatego też, jeśli

masz klasę pochodną o nazwie `MyArray` dziedziczącą po wbudowanym typie `Array`, metoda, taka jak `slice()`, zwróci egzemplarz typu `MyArray`. Spójrz na poniższy fragment kodu.

```
class MyArray extends Array {  
    // Pusta definicja.  
}  
  
let items = new MyArray(1, 2, 3, 4),  
    subitems = items.slice(1, 3);  
  
console.log(items instanceof MyArray);    // Wartość true.  
console.log(subitems instanceof MyArray); // Wartość true.
```

W powyższym fragmencie kodu metoda `slice()` zwraca egzemplarz `MyArray`. Metoda `slice()` jest dziedziczona po `Array` i normalnie zwraca egzemplarz typu `Array`. Zmiana jest w rzeczywistości przeprowadzana w tle przez właściwość `Symbol.species`.

Właściwość `Symbol.species` jest doskonale znana i używana do zdefiniowania statycznej właściwości akcesora zwracającej funkcję. Ta funkcja będzie konstruktorem przeznaczonym do użycia, gdy egzemplarz klasy ma zostać utworzony w metodzie egzemplarza (zamiast wykorzystania konstruktora). Wymienione poniżej wbudowane typy danych mają zdefiniowaną właściwość `Symbol.species`:

- `Array`;
- `ArrayBuffer` (jego dokładne omówienie znajdziesz w rozdziale 10.);
- `Map`;
- `Promise`;
- `RegExp`;
- `Set`;
- typowane tablice (jego dokładne omówienie znajdziesz w rozdziale 10.).

Wszystkie typy wymienione na powyższej liście mają właściwość `Symbol.species` zwracającą wiązanie `this`, co oznacza, że właściwość ta zawsze będzie zwracała funkcję konstruktora. Gdyby tę funkcjonalność zaimplementować w klasie niestandardowej, wówczas kod mógłby przedstawiać się następująco.

```
// Kilka wbudowanych typów danych używa rozwiązania podobnego do poniższego.  
class MyClass {  
    static get [Symbol.species]() {  
        return this;  
    }  
  
    constructor(value) {  
        this.value = value;  
    }  
}
```

```

    }

    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}

```

W powyższym fragmencie kodu doskonale znany `Symbol.species` został użyty w celu przypisania statycznej właściwości akcesora do `MyClass`. Zwróć uwagę na obecność metody getter bez setter, ponieważ zmiana typu klasy jest niemożliwa. Każde wywołanie `this.constructor[Symbol.species]` zwróci wartość `MyClass`. Metoda `clone()` używa tej definicji w celu zwrotu nowego egzemplarza zamiast bezpośredniego użycia `MyClass`, co spowodowałoby nadpisanie tej wartości przez klasę pochodną. Spójrz na poniższy fragment kodu.

```

class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
    }
    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}

class MyDerivedClass1 extends MyClass {
    // Pusta definicja.
}

class MyDerivedClass2 extends MyClass {
    static get [Symbol.species]() {
        return MyClass;
    }
}

let instance1 = new MyDerivedClass1("foo"),
    clone1 = instance1.clone(),
    instance2 = new MyDerivedClass2("bar"),
    clone2 = instance2.clone();

console.log(clone1 instanceof MyClass);           // Wartość true.
console.log(clone1 instanceof MyDerivedClass1);   // Wartość true.
console.log(clone2 instanceof MyClass);           // Wartość true.
console.log(clone2 instanceof MyDerivedClass2);   // Wartość false.

```

W omawianym przykładzie `MyDerivedClass1` dziedziczy po `MyClass` i nie zmienia właściwości `Symbol.species`. Po wywołaniu metody `clone()` zwracany jest egzemplarz `MyDerivedClass1`, ponieważ `this.constructor[Symbol.species]` zwraca `MyDerivedClass1`. Klasa `MyDerivedClass2` dziedziczy po `MyClass` i nadpisuje `Symbol.species` w celu zwrotu `MyClass`. Po wywołaniu metody `clone()` w egzemplarzu `MyDerivedClass2` wartością zwrótną jest egzemplarz `MyClass`. Za pomocą `Symbol.species` i dowolnej klasy pochodnej można ustalić typ wartości, jaka powinna być zwrócona, gdy metoda zwraca egzemplarz.

Przykładowo klasa `Array` używa `Symbol.species` do wskazania klasy przeznaczonej do wykorzystania przez metodę zwracającą tablicę. W klasie pochodnej typu `Array` można ustalić typ obiektu zwracanego przez metody odziedziczone, np. tak jak pokazałem poniżej.

```
class MyArray extends Array {
  static get [Symbol.species]() {
    return Array;
  }
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);           // Wartość true.
console.log(subitems instanceof Array);          // Wartość true.
console.log(subitems instanceof MyArray);        // Wartość false.
```

W powyższym fragmencie kodu nadpisujemy `Symbol.species` w egzemplarzu `MyArray` dziedziczącym po `Array`. Wszystkie metody odziedziczone zwracające tablicę będą teraz używały egzemplarza `Array` zamiast `MyArray`.

Ogólnie rzecz biorąc, powinieneś stosować właściwość `Symbol.species` zawsze wtedy, gdy chcesz użyć `this.constructor` w metodzie klasy. To pozwala klasom pochodnym na łatwe nadpisanie zwracanego typu. Ponadto jeśli stworzysz klasy pochodne na podstawie klasy zawierającej `Symbol.species`, upewnij się, że użyłeś tej wartości zamiast konstruktora.

Użycie właściwości `new.target` w konstruktorze klasy

W rozdziale 3. poznałeś właściwość `new.target` oraz dowiedziałeś się, jak jej wartość ulega zmianie w zależności od sposobu wywołania funkcji. Istnieje również możliwość użycia właściwości `new.target` w konstruktorze klasy w celu ustalenia, jak klasa jest wywoływana. W tym prostym przykładzie właściwość `new.target` jest odpowiednikiem funkcji konstruktora klasy, co pokazałem poniżej.

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    this.length = length;
    this.width = width;
  }
}

// Właściwość new.target ma wartość Rectangle.
var obj = new Rectangle(3, 4);      // Dane wyjściowe to true.
```

W omawianym przykładzie pokazałem, że właściwość `new.target` jest odpowiednikiem `Rectangle` podczas wywołania `new Rectangle(3, 4)`. Konstruktor klasy nie może być wywołany bez `new`, więc właściwość `new.target` zawsze będzie zdefiniowana w konstruktorze klasy. Jednak jej wartość nie zawsze będzie taka sama. Spójrz na poniższy fragment kodu.

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    this.length = length;
    this.width = width;
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, length)
  }
}

// Właściwość new.target ma wartość Square.
var obj = new Square(3);      // Dane wyjściowe to false.
```

Egzemplarz `Square` wywołuje konstruktor `Rectangle`, więc podczas wywołania konstruktora `Rectangle` wartością właściwości `new.target` jest `Square`. To bardzo ważny aspekt, ponieważ daje każdemu konstruktorowi możliwość zmiany zachowania w zależności od sposobu jego wywołania. Przykładowo abstrakcyjną klasę bazową (niemożliwą do bezpośredniego wywołania) można za pomocą właściwości `new.target` zdefiniować w poniższy sposób.

```
// Abstrakcyjna klasa bazowa.
class Shape {
  constructor() {
    if (new.target === Shape) {
      throw new Error("Nie można bezpośrednio utworzyć egzemplarzy
        ↳ tej klasy.")
    }
  }
}
```

```

    }
  }
}

class Rectangle extends Shape {
  constructor(length, width) {
    super();
    this.length = length;
    this.width = width;
  }
}

var x = new Shape(); // Następuje zgłoszenie błędu.

var y = new Rectangle(3, 4); // Bez błędu.
console.log(y instanceof Shape); // Wartość true.

```

W omawianym przykładzie konstruktor klasy Shape zgłosi błąd, gdy wartością `new.target` będzie Shape, co oznacza, że wywołanie `new Shape()` zawsze zgłosi błąd. Jednak nadal można używać Shape jako klasy bazowej i takie podejście zastosowaliśmy w przypadku egzemplarza Rectangle. Wywołanie `super()` powoduje wykonanie konstruktora Shape, a wartością `new.target` jest Rectangle, więc konstruktor bez problemów kontynuuje działanie.

UWAGA *Ponieważ klasa nie może być wywołana bez słowa kluczowego `new`, właściwość `new.target` w konstruktorze klasy nigdy nie będzie miała wartości `undefined`.*

Podsumowanie

Klasy w specyfikacji ECMAScript 6 ułatwiają użycie dziedziczenia w języku JavaScript, a programista może wykorzystać posiadane doświadczenie dotyczące dziedziczenia poznanego w innych językach programowania. Na początku klasy w specyfikacji ECMAScript 6 stanowiły syntaktyczny dodatek dla modelu klasycznego dziedziczenia w ECMAScript 5, ale później dodano do nich wiele funkcji, aby zmniejszyć poziom popełnianych błędów.

Klasy w specyfikacji ECMAScript 6 działają z dziedziczeniem prototypowym przez zdefiniowanie niestatycznych metod w prototypie klasy, natomiast metody statyczne są umieszczane w konstruktorze. Wszystkie metody nie poddają się wyliczeniu, co lepiej odpowiada zachowaniu wbudowanych obiektów, których metody domyślnie zwykle nie poddają się wyliczeniu. Ponadto konstruktor klasy nie może być wywołany bez słowa kluczowego `new`, co chroni przed przypadkowym wywołaniem klasy jako funkcji.

Dziedziczenie oparte na klasie pozwala na utworzenie klasy pochodnej na podstawie innej klasy, funkcji lub wyrażenia. Oznacza to możliwość wywołania funkcji w celu ustalenia prawidłowej bazy, po której będzie odbywać się dziedziczenie.

Podczas tworzenia nowej klasy możesz więc użyć domieszek oraz innych wzorców. Teraz dziedziczenie po wbudowanych typach danych, np. `Array`, działa zgodnie z oczekiwaniami.

Właściwość `new.target` w konstruktorze klasy może zachowywać się odmiennie, w zależności od sposobu wywołania klasy. Najczęściej możliwość tę wykorzystuje się do utworzenia abstrakcyjnej klasy bazowej, która zgłasza błąd podczas próby bezpośredniego utworzenia jej egzemplarza, choć nadal pozwala na dziedziczenie poprzez inne klasy.

Ogólnie rzecz biorąc, klasy są ważnym dodatkiem do języka JavaScript. Oferują znacznie zwiększoną składnię oraz lepszą funkcjonalność i pozwalają na tworzenie obiektów niestandardowych typów w bezpieczny i spójny sposób.

Skorowidz

A

- akcesor, 216
- API refleksji, 297, 299
- asynchroniczne
 - wczytywanie modułu, 357
 - wykonywanie zadania, 199, 291

B

- bezpieczne liczby całkowite, 362
- błąd składni, 179
- błędy, 282
 - funkcji executor, 275
- BMP, basic multilingual plane, 36
- bufor tablicy, 247

C

- ciągi tekstowe, 35
- CORS, 359
- cykl życiowy obietnicy, 268

D

- dane obiektu prywatnego, 170
- definicja
 - metody, 110
 - tagu, 55
- const, 25, 26
- deklaracja
 - const w pętli, 31
 - hoisting, 21

- klasy, 208
- let, 23, 25
- let w pętli, 30
- var, 21

- deklaracje na poziomie bloku, 23
- delegowanie generatora, 197, 206
- deskryptor
 - obiektu, 315
 - właściwości, 313
- destrukuryzacja, 113
 - mieszana, 125
 - obiektu, 114
 - parametrów, 125
 - przypisania, 115, 121
 - tablicy, 120
 - zagnieżdżonego obiektu, 118
 - zagnieżdżonej tablicy, 123
- domieszka, 101
- domyślne iteratory, 187
- dostęp
 - do danych, 113
 - do domyślnego iteratora, 182
 - do prototypu, 107
- DSL, domain-specific language, 50
- dziedziczenie, 221, 224, 228
 - po obietnicach, 289

E

- ECMAScript 6, 15
- eksport, 344
 - wartości domyślnej, 350
 - wiązania, 352

- element script, 355
- elementy
 - poddające się iteracji, 181, 191, 242
 - resztowe, 123
- executor, 270

F

- formalizacja właściwości `__proto__`, 365
- formalna definicja metody, 110
- formatowanie podstawowe ciągu tekstowego, 50
- funkcja
 - executor, 275
 - strzałki, 81, 88
 - w pętli, 29
- funkcje, 59
 - identyfikowanie, 74
 - konstruktor Function, 72
 - na poziomie bloku, 79, 81
 - nienazwane parametry, 68
 - parametry domyślne, 59
 - pierwszoklasowe, 214
 - podwójne przeznaczenie, 76
 - proxy, 319
 - wywoływanie, 77
- funkcjonalność iteratorów, 192

G

- generator, 177, 206
 - delegowanie, 197
 - metody, 218
- generowane nazwy
 - elementów składowych, 217
 - właściwości, 98

I

- identyfikacja
 - funkcji strzałki, 88
 - liczby całkowitej, 362
 - podciągów tekstowych, 42
 - tablic, 146
- identyfikatory Unicode, 364
- IIFE, immediately invoked function expression,
 - 84, 171
- implementacja klasy MyArray, 332
- import, 345
 - bez wiązań, 353
 - całego modułu, 347

- pojedynczego wiązania, 346
- przestrzeni nazw, 347
- wartości domyślnej, 351
- wielu wiązań, 346
- inicjalizacja
 - mapy, 166
 - słabej mapy, 169
 - właściwości, 96
- iterator, 176, 205
 - entries(), 184
 - keys(), 186
 - values(), 185
- iteratory
 - ciągu tekstowego, 189
 - kolekcji, 184
 - NodeList, 190
 - przekazywanie argumentów, 192
 - wbudowane, 184
 - zaawansowana funkcjonalność, 192
 - zgłaszanie błędu, 194

K

- kategorie obiektu, 95
- klasa, 207
 - MyArray, 332
- klasy
 - pochodne, 221, 225
 - wbudowane, 228
- koercja symbolu, 135
- kolejka zadań, 264
- kolejność operacji, 370
- koncepcja TDZ, 66
- konstruktor
 - Function, 72
 - zbioru, 158
- konstruktory klasy, 232
 - bez operatora new, 322
 - możliwe do wywołania, 325
- konstruowanie pułapek, 319
- kontekst programowania asynchronicznego, 264
- konwersja tablicy na zbiór, 161
- kształt obiektu, 302

L

- liczbowe typy danych, 246
- liczby całkowite, 361
 - bezpieczne, 362
 - identyfikacja, 362
- literal obiektu, 96, 103

Ł

łańcuch obietnic, 283, 284
łączenie obietnic, 281

M

mapowanie konwersji, 241
mapy, 153, 164
 inicjalizacja, 166
 metody, 165
 słabe, 168, 172
metawłaściwość `new.target`, 78
metoda
 `Array.from()`, 239, 242
 `Array.of()`, 238
 `Array.prototype.includes()`, 371
 `clear()`, 165
 `codePointAt()`, 37
 `copyWithin()`, 245
 `defineProperty()`, 317
 `delete()`, 165
 `fill()`, 244
 `find()`, 243
 `findIndex()`, 243
 `forEach()`, 159, 167, 173
 `from()`, 257
 `getOwnPropertyDescriptor()`, 317
 `has()`, 165
 `normalize()`, 38
 `Object.assign()`, 101, 103, 111
 `Object.defineProperty()`, 151, 314
 `Object.getPrototypeOfNames()`, 151
 `Object.is()`, 100
 `Object.keys()`, 151
 `Object.prototype.toString()`, 147
 `of()`, 257
 `Promise.all()`, 287
 `Promise.race()`, 288
 `Promise.reject()`, 273
 `Promise.resolve()`, 273
 `repeat()`, 44
 `String.fromCodePoint()`, 38
 `Symbol.hasInstance`, 138
 `Symbol.toPrimitive`, 144
metody
 deskryptora, 316
 generatora, 218
 mapy, 165
 obiektu generatora, 180

 obiektu `Math`, 363
 słabej mapy, 169
 tablic, 256
model zdarzeń, 264
moduł, 344

N

nadpisanie konstruktora, 323
najlepsze praktyki
 wiązanie bloku, 33
natychmiast wykonywane wyrażenie funkcji,
 IIFE, 84
nazwy
 elementów, 349
 elementów składowych, 217
 importowane, 350
 lokalne, 349
 prywatne, 131
 stałych, 211
 właściwości, 98
nienazwane parametry, 68

O

obejścia, 155
obiekt
 arguments, 71
 egzotyczny, 96
 `Math`, 363
 standardowy, 96
 tablicy, 298
 thenable, 274
 wbudowany, 96
 zwykły, 96
obiekty poddające się iteracji, 183
obietnice, 263, 267
 asynchroniczne wykonywanie zadania, 291
 cykl życiowy, 268
 dziedziczenie, 289
 łączenie, 281
 nierozstrzygnięte, 270
 obiekt thenable, 274
 oczekiwanie, 295
 odrzućcie, 295
 procedura obsługi odrzucenia, 276
 spełnione, 273, 295
 udzielanie odpowiedzi, 287
 zwrot, 284

- obsługa odrzucenia obietnicy, 276–279
- odpowiednik kanoniczny, 38
- odwołanie super, 107
- ograniczenia
 - deskryptora obiektu, 315
 - słabych map, 172
 - operandu, 370
- operacje
 - eksportu, 344
 - importu, 345
- operator
 - new, 208, 322
 - rozszczepiania, 72, 191
 - wykładniczy, 370
- optymalizacja wywołania ogonowego, 92

P

- parametry domyślne, 59
 - koncepcja TDZ, 66
 - symulowanie wartości, 60
 - wyrażenia, 64
- parametry nienazwane, 68
- parametry resztowe, 69
 - ograniczenia, 70
- pary surogatów, 36
- pętla, 29, 175
 - for-in, 30
 - for-of, 30, 181, 206
 - zdarzeń, 264
- piekło wywołań zwrotnych, 266
- podstawowa przestrzeń wielojęzyczna, BMP, 36
- podwójne przeznaczenie funkcji, 76
- polecenie
 - import, 348
 - return, 196
 - yield, 179, 200
- ponowny eksport wiązania, 352
- porównywanie wartości, 372
- powielenie
 - metod deskryptora, 316
 - właściwości literału obiektu, 103
- problem tablicy, 298, 327
- programowanie asynchroniczne, 264
- prosta lista parametrów, 374
- prototyp, 105
 - pułapka get, 335
 - pułapka has, 337
 - pułapka set, 336

- proxy, 297, 298, 319
 - jako prototyp, 334, 338
 - możliwe do odwołania, 326
- przechwytywanie błędów, 282
- przekazanie argumentów do iteratora, 192
- przesłanianie metod klasy, 224
- przestrzeń dodatkowa, 36
- pułapka
 - deleteProperty, 305
 - get, 302, 335
 - has, 304, 337
 - ownKeys, 318
 - set, 300, 336
- pułapki
 - deskryptora właściwości, 313
 - prototypu proxy, 307, 308
 - związane z rozbudową obiektu, 311
- punkty kodowe, 36

R

- refleksje, 298
- rozbudowa obiektu, 311

S

- sekwencja wczytywania modułu, 356
- service worker, 358
- silny zbiór, 162
- składnia funkcji strzałki, 82
- słabe mapy, 168
- słaby zbiór, 162
- słowo kluczowe
 - class, 213
 - default, 350
 - export, 344
 - function, 111
 - import, 345
- specyfikator modułu, 346, 359
- stała, 25
- statyczne elementy składowe, 220, 224
- symbol Symbol.iterator, 205
- symbole, 131
 - koercja, 135
 - powszechnie znane, 137
 - tworzenie, 132
 - udostępnienie wewnętrznych operacji, 137
 - użycie, 133
 - właściwości, 136
 - współdzielenie, 134

symulowanie wartości parametrów domyślnych, 60
szablon z tagiem, 54
szablony literalów, 50, 57
 składnia podstawowa, 50
 wielowierszowy ciąg tekstowy, 51

T

tablice, 88, 237, 255
 brakujące metody, 259
 dodawanie elementu, 328
 metody, 243, 256
 metody dodatkowe, 259
 odczyt danych, 249
 podobieństwa, 255
 różnice, 257
 różnice behawioralne, 258
 tworzenie, 237
 typowane, 246, 252, 255
 usuwanie elementów, 330
 wykrywanie indeksu, 328
 zapis danych, 249
tag, 55
TDZ, temporal dead zone, 27
tryb ścisły, 373
tworzenie
 harmonogramu zadań, 271
 nierozstrzygniętej obietnicy, 270
 proxy, 300
 słabego zbioru, 162
 spełnionej obietnicy, 273
 symbolu, 132
 tablicy, 237
 zbioru, 156
tymczasowo martwa strefa, TDZ, 27
typ
 NodeList, 190
 pierwszoklasowy, 214
 wyliczeniowy, 104
typowane tablice, 246, 252
typy
 tablic, 255
 zbiorów, 163

U

uchwyt, 300
ukrycie istnienia właściwości, 304
Unicode, 35
uniemożliwienie usunięcia właściwości, 305

UTF-16, 36
użycie
 funkcji na poziomie bloku, 80
 metody Array.from(), 242
 metody Promise.reject(), 273
 metody Promise.resolve(), 273
 modułu, 354, 355
 opcji u, 42
 proxy jako prototypu, 334
 pułapki get, 335
 pułapki has, 337
 pułapki set, 336
 słabych map, 168
 symbolu, 133
 właściwości new.target, 232

W

wartości parametrów domyślnych, 61, 62
wartość domyślna
 eksport, 350
 import, 351
 w module, 350
wartość zwrotna, 283
 iteratora, 176
wbudowane iteratory, 184
wczytywanie modułu, 354, 356
 asynchroniczne, 357
 w wątku roboczym, 358
web worker, 358
weryfikacja
 kształtu obiektu, 302
 parametrów funkcji, 320
 właściwości, 300
wiązanie
 arguments, 88
 bloku, 21, 33
 bloku globalnego, 32
 bloku w pętli, 28
 this, 85
widok, 248, 249, 252
 zwrot informacji, 249
 charakterystyczny dla typu, 253
wielkość elementu, 255
wielowierszowy ciąg tekstowy, 50
własne właściwości, 209
właściwości
 akcesora, 103, 216
 literalu obiektu, 103
 symbolu, 136
 typu wyliczeniowego, 104

- właściwość
 - __proto__, 365
 - flags, 49
 - name, 74
 - new.target, 232
 - Symbol.isConcatSpreadable, 140
 - Symbol.match, 142
 - Symbol.replace, 142
 - Symbol.search, 142
 - Symbol.species, 229
 - Symbol.split, 142
 - Symbol.toStringTag, 146
 - Symbol.unscopables, 150
 - System.iterator, 181
- współdzielenie symboli, 134
- wyciek pamięci, 162
- wykrywanie indeksu tablicy, 328
- wyrażenia
 - klasy, 212
 - nazwanych klas, 213
 - parametru domyślnego, 64
- wyrażenia regularne, 45
 - opcja y, 45
 - powielanie, 48
 - właściwość flags, 49

- wywołania zwrotne, 265
- wywołanie
 - konstruktora, 322
 - ogonowe, 89
- wzorzec wywołania zwrotnego, 265

Z

- zadania asynchroniczne, 294
- zasięg
 - bloku, 23
 - leksykalny, 23
- zastępowanie ciągu tekstowego, 53
- zbiór, 153, 157
 - silny, 162
 - słaby, 162
 - usuwanie elementów, 158
- zgłaszanie błędu w iteratorze, 194
- zliczanie punktów kodowych, 41
- zmiana
 - nazwy elementu, 349
 - prototypu obiektu, 105
- zmienne lokalne, 117
- znaki sterujące HTML, 50
- związłe metody, 97
- zwrot obietnicy, 284

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 



JAVASCRIPT TO DOJRZAŁOŚĆ, ELASTYCZNOŚĆ I NAJNOWSZE STANDARDY!

Specyfikacja ECMAScript 6 jest najważniejszym uaktualnieniem w dotychczasowej historii języka JavaScript. Zrozumienie jej ma kluczowe znaczenie dla wszystkich programistów JavaScriptu. Tworzy ona solidny fundament i to właśnie na nim będą budowane wszystkie aplikacje JavaScriptu w przyszłości.

Oto podręcznik przeznaczony dla średnio zaawansowanych i zaawansowanych programistów JavaScriptu, którzy korzystają ze środowiska przeglądarki WWW lub Node.js. Omówiono tu zagadnienia wiązania bloków, ciągów tekstowych, wyrażeń regularnych, a także zmiany wprowadzone w funkcjach. Przedstawiono pełne wprowadzenie do typów obiektów oraz składni, które pojawiły się w JavaScriptcie wraz ze specyfikacją ECMAScript 6. Nie zabrakło przykładów kodu działającego w dowolnym środowisku JavaScriptu. Dodatkowo zaprezentowano zmiany wprowadzone wraz z najnowszym standardem ECMAScript 7 (2016).

W książce omówiono między innymi:

- natywne tablice JavaScriptu i ich nowe możliwości
- obietnice i programowanie asynchroniczne
- API refleksji
- wykorzystanie proxy do kontroli obiektów
- hermetyzację kodu za pomocą modułów

Nicholas C. Zakas — pisze aplikacje internetowe od niemal dwudziestu lat. Jest doskonale znanym i uznanym ekspertem w dziedzinie tworzenia front-endu; przyczynia się do kształtowania najlepszych praktyk w tym zakresie. Specjalizuje się w stosowaniu takich technik jak JavaScript, Dynamic HTML, CSS, XML oraz XSLT. Kilka lat pracował w firmie Yahoo!, w której pełnił funkcję naczelnego inżyniera do spraw związanych z jej główną witryną. Jest autorem wielu książek dotyczących technik programistycznych.

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-3403-8



9 788328 334038

cena: 67,00 zł

Informatyka w najlepszym wydaniu

